임베디드 리눅스 커널 프로그래밍

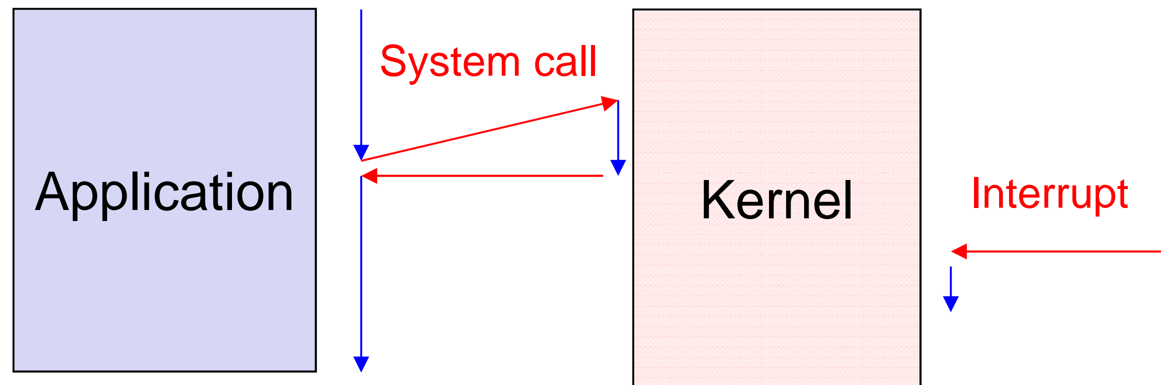
Kernel Programming 이란?

- 커널 모드에서 수행하는 프로그램을 작성하는 것

- 커널 프로그래밍 종류
  - Linux kernel core 기능 추가
  - Linux kernel 알고리즘 개선
  - Linux kernel 모듈 프로그래밍 – 커널 컴파일 필요 없음

Kernel Program vs. Application Program (1)

- 수행 방법
  - Application Program: 처음부터 순차적으로 수행
  - Kernel: 응용프로그램이 호출한 **system call**이나 **interrupt handler**를 수행하기 위해 비동기적으로 수행



Kernel Program vs. Application Program (2)

- Library
  - Application: 모든 library를 link하고 사용할 수 있다.
  - Kernel: kernel에서 export 하는 것들만 사용할 수 있다.

    EXPORT_SYMBOL(func_name)

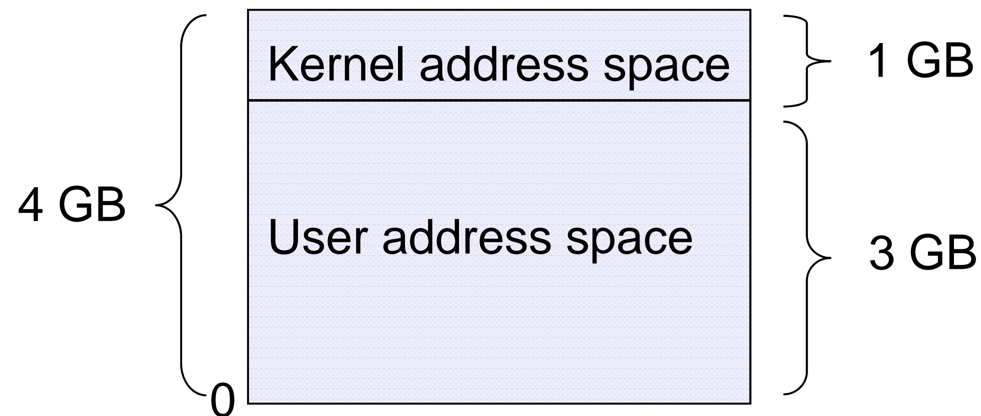
- Kernel mode vs User mode
  - Application: user mode에서 수행되며 하드웨어에 직접 접근하는 것과 메모리에 대한 허용되지 않은 접근이 제한된다.
  - Kernel: kernel mode에서 수행되며 모든 것이 허용된다.

Kernel Program vs. Application Program (3)

Address Space

- Application Program과 Kernel Program은 서로 다른 memory address space를 가지고 있음

32-bit virtual address space에서



Kernel Program vs. Application Program(4)

- Namespace pollution 문제
  - Application Program:
    - 현재 개발하는 프로그램에서만 각 함수와 변수의 이름을 구별하여 주면 된다.
  - Kernel Program:
    - 리눅스 커널 = 커널 core + 다수의 커널 모듈
    - 현재 개발하는 커널 모듈 뿐 만 아니라 커널 전체에서 함수와 변수의 이름이 충돌하지 않도록 해야 한다.

Kernel programming 주의 사항

- Library
  - stdio.h와 같이 보통 프로그램에서 사용하는 헤더 파일을 include해서 사용할 수 없음
  - 다음에 선언된 헤더파일 만 include 함
    - /usr/include/linux 와 /usr/include/asm

      (예) #include <linux/***.h >
      #include <asm/***.h>
- Namespace pollution
  - 외부 파일와 link하지 않을 모든 심볼을 static으로 선언함.
    - **static** int name;
  - 외부 파일과 link할 symbol을 symbol table 등록
    - **EXPORT_SYMBOL**(name);   심볼 name을 export함
  - 전역 변수는 잘 정의된 prefix를 붙여 준다.
    - (ex) sys_open()

Kernel programming 주의 사항(2)

- Fault handling
  - Kernel 은 하드웨어 접근에 대해 어떠한 제한도 없기 때문에 커널에서의 에러는 시스템에 치명적인 결과를 발생시킨다.
  - 함수 호출 등의 작업 시 모든 에러 코드를 검사하고 처리해야 한다.
- Address space
  - 커널이 사용하는 stack의 크기는 제한되어 있고, 인터럽트 핸들러도 같은 스택을 사용할 수도 있으므로 큰 배열을 지역변수로 사용하거나, recursion이 많이 일어나지 않도록 주의해야 한다.
  - Application과 data를 주고 받기 위해 특별한 함수를 사용하여야 한다.

    → call by reference
    - (뒷부분에서 이를 위한 몇 가지 함수를 소개한다.)
- 기타
  - 실수연산이나 MMX/SSE 연산(x86)을 사용할 수 없다.

커널 Data Type

- C언어의 원래의 자료형 대신에 typedef으로 정의된 자료형이 많이 사용된다.

- 커널에서 여러 종류의 값을 나타내는 데에 int 나 long 등 대신에 접미사 _t로 끝나는 자료형을 많이 사용한다.
  - pid_t, uid_t, gid_t, dev_t, size_t, ...
  - <linux/types.h> 에 정의되어 있음

- 데이터의 크기를 명시적으로 표현한 자료형
  - u8, u16, u32, s8, s16, s32 (커널 코드에서만 사용 가능)
  - 사용자 프로그램에서는 __u8, __s8과 같이 _를 두 개 붙여서 사용
  - <asm/types.h>에 정의되어 있음

Kernel Interface 함수

- 주의 사항
  - Kernel program은 일반적인 library를 사용하지 못하고 kernel에서 export한 함수들 만을 사용할 수 있다.
- Kernel interface 함수 분류
  - I/O port, I/O memory
  - Interrupt
  - Memory
  - Synchronization
  - Kernel message 출력
  - Device Driver register

Kernel Interface 함수 – port I/O

- I/O에 대한 가상주소 맵핑
  - void *ioremap(unsigned long phys_addr, unsigned long size);
  - void *ioremap_nocache(unsigned long phys_addr, unsigned long size);
  - void iounmap(void * addr);
- I/O memory 읽기 — old version
  - unsigned int ioread8(void *addr); — readb(addr)
  - unsigned int ioread16(void *addr); — readw(addr)
  - unsigned int ioread32(void *addr); — readl(addr)
- I/O memory 쓰기
  - void iowrite8(u8 value, void *addr); — writeb(value, addr)
  - void iowrite16(u16 value, void *addr); — writew(value, addr)
  - void iowrite32(u32 value, void *addr); — writel(value, addr)
- I/O port 입출력 (x86)
  - unsigned inb(unsigned port); / inw() / inl()
  - void outb(unsigned char byte, unsigned port) / outw() / outl()

Kernel Interface 함수 - Interrupt

- 인터럽트 비활성화/활성화 (macro)
  - local_irq_disable()
  - local_irq_enable()
- 인터럽트 비활성화/활성화시 status register 저장/복원 (macro)

  unsigned long flags;
  - local_irq_save(flags)
  - local_irq_restore(flags)
- 인터럽트 핸들러 등록/해제
  - int request_irq(unsigned int **irq**,
    irqreturn_t (***handler**)(int, void *, struct pt_regs *),
    unsigned long **irqflags**,
    const char ***devname**,
    void* **device**)
  - void free_irq(unsigned int **irq, void *devid**)
    - request_irq()에서 획득한 irq를 반납함

Kernel Interface 함수 - Memory

- 커널 물리적 메모리 동적 할당/해제
  - void *kmalloc(size_t size, int flags)
    - 커널 메모리 할당: 최대크기 제한 있음(초기128KB, 현재 증가함)
    - 물리적 주소 공간에서 연속적인 메모리를 할당한다.
    - flags: GFP_USER(사용자 프로그램용 메모리 할당),
GFP_KERNEL(커널용 메모리 할당, sleep가능),
GFP_ATOMIC(인터럽트 핸들러 등에서 사용, sleep불가) 등
  - void kfree(void *obj)
- 커널 가상메모리 동적 할당/해제
  - void * vmalloc(unsigned int len)
    - 커널 메모리 할당, 크기 제한 없음
    - 가상 주소 공간에서 연속적인 메모리 영역을 할당
  - void vmfree(void *addr)
    - vmalloc()에서 할당받은 커널 메모리를 반납

Kernel Interface 함수 – 데이터 복사

- 사용자 공간과 커널 공간 사이에 데이터 복사
  - unsigned long copy_from_user(void *to, const void *from, unsigned long n).
  - unsigned long copy_to_user(void *to, const void *from, unsigned long n)
  - put_user(data, ptr) / get_user(ptr) (macro)
    - ptr이 가리키는 사용자 공간에 data를 전달(put)하거나 가져옴(get)
    - ptr의 type에 따라서 복사할 데이터 크기를 인식함(1,2,4 byte)
  - 커널 공간에서 사용자 공간의 메모리를 읽고 쓰는 데 접근할 수 있는 지를 검사하는 동작을 포함

- 메모리 값 설정
  - void * memset(void *s, char c, size_t count)
    - 메모리 s에 c를 count만큼 복사

Kernel Interface 함수 – Synchronization

- 동기화
  - void sleep_on(struct wait_queue **q) // deprecate
  - void sleep_in_interruptible(struct wait_queue **q) // deprecate
    - q의 번지를 event로 sleep하며, uninterruptible/interruptible
  - wait_event(queue, condition) (macro)
  - wait_event_interruptible(queue, condition)
    - condition이 1일 때까지 wait, queue번지를 event로 하여 sleep하며 uninterruptible/interruptible
  - void wake_up(struct wait_queue **q)
  - void wake_up_interruptible(struct wait_queue **q)
    - sleep_on ... (q) / wait_event ...(q, cond)에 의해 sleep한 task를 wakeup

Kernel Interface 함수 – 메시지 출력

- 표준 출력
  - printk(const char *fmt,… .)
    - printf의 커널 버전
    - printk(LOG_LEVEL message)

    - LOG_LEVEL:KERN_EMERG, KERN_ALERT, KERN_ERR,
      KERN_WARNING, KER_INFO, KERN_DEBUG
      <linux/kernel.h>참조
  - 예
    - printk("<1>Hello, World");
    - printk(KERN_WARNING "warning… \n"); // <4>

Kernel Interface 함수 – device driver 관련

- Device Driver 등록/해제
  - int register_xxxdev(unsigned int major, const char *name, struct file_operations *fops)
    - character/block driver를 xxxdev[major]에 등록
    - xxx: chr 또는 blk
  - int unregister_xxxdev(unsigned int major, const char *name)
    - xxxdevs[major]에 등록되어있는 device driver를 제거
  - int register_netdev(struct net_device *dev)
  - int unregister_netdev(struct net_device *dev)
  - MAJOR(kdev_t dev), MINOR(kdev_t dev)
    - 장치번호dev로부터 major/minor 번호를 구함

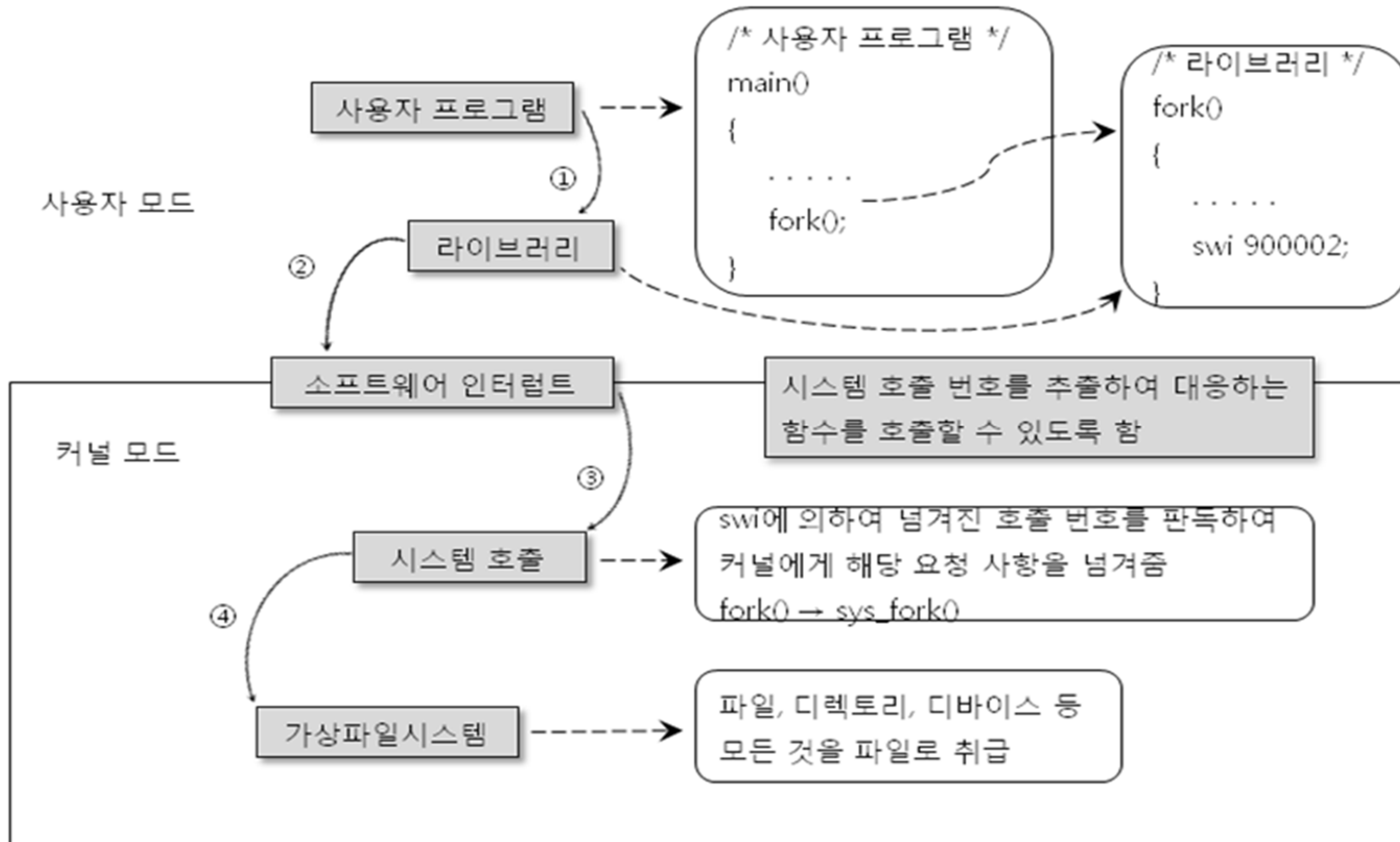
Kernel Interface 함수 – signal 함수

- 디바이스 드라이버에서 사용자 프로그램에 signal을 보내는 함수
  - int kill_proc(pid_t pid, int signum, int priv)

- pid
  - current : 현재 프로세스 task_struct에 대한 ptr (macro)
  - current->pid : 현재 프로세스의 pid
  - current->ppid : 현재 프로세스의 부모 프로세스의 pid

- 사용자 프로그램에서 signal handler 등록
  - void (*signal(int signum, void(*handler)(int)) (int);

시스템 호출 과정



시스템 호출의 구현

- 시스템 호출 함수 정의
  - **sys_**funcname ... 사용자 프로그램에서는 funcname호출
- 시스템 호출 번호 할당
  - include/asm-arm/unistd.h 파일에 추가
- 시스템 호출 함수 등록
  - arch/arm/kernel/call.S 파일에 등록
- 커널 재 컴파일

시스템 호출 함수 정의

- kernel 소스의 kernel 디렉토리에
  시스템 호출 함수를 정의한 프로그램 파일 (hello.c) 생성

```
#include <linux/kernel.h>

asmlinkage void sys_hello()
{
    printk ("Hello, this is new system call!\n");
}
```

asmlinkage : stack을 통하여 parameter 전달
fastcall : 몇개의 parameter는 register를 통하여 전달

시스템 호출 번호 할당

- include/asm-arm/unistd.h 파일 편집

```
#include <linux/linkage.h>

#if defined(__thumb__) || defined(__ARM_EABI__) && !defined(__USE_ARM_OABI_SYSCA
LLS)
#define __NR_SYSCALL_BASE       0
#else
#define __NR_SYSCALL_BASE       0x900000
#endif

/*
 * This file contains the system call numbers.
 */

#define __NR_restart_syscall            (__NR_SYSCALL_BASE+  0)
#define __NR_exit                       (__NR_SYSCALL_BASE+  1)
#define __NR_fork                       (__NR_SYSCALL_BASE+  2)
#define __NR_read                       (__NR_SYSCALL_BASE+  3)
#define __NR_write                      (__NR_SYSCALL_BASE+  4)
#define __NR_open                       (__NR_SYSCALL_BASE+  5)
#define __NR_close                      (__NR_SYSCALL_BASE+  6)
                                        /* 7 was sys_waitpid */
#define __NR_creat                      (__NR_SYSCALL_BASE+  8)
#define __NR_link                       (__NR_SYSCALL_BASE+  9)
```

.....

fork →
SWI 0x900002

- unistd.h 파일의 시스템 호출번호의 마지막에 sys_hello()를 위한 번호 322를 추가함

```
#define __NR_get_mempolicy          (__NR_SYSCALL_BASE+320)
#define __NR_set_mempolicy          (__NR_SYSCALL_BASE+321)
#define __NR_hello
   (__NR_SYSCALL_BASE+322)
```

  - sys_hello 함수는 SWI 0x900322 로 호출되어 수행됨

시스템 호출 함수 등록

- linux/arch/arm/kernel/calls.S 파일 편집

```
#ifndef NR_syscalls
#define NR_syscalls 328
#else

100:
/* 0 */          .long   sys_restart_syscall
                 .long   sys_exit
                 .long   sys_fork_wrapper
                 .long   sys_read
                 .long   sys_write
/* 5 */          .long   sys_open
                 .long   sys_close
                 .long   sys_ni_syscall          /* was sys_waitpid */
                 .long   sys_creat
                 .long   sys_link
/* 10 */         .long   sys_unlink
                 .long   sys_execve_wrapper
                 .long   sys_chdir
                 .long   sys_time        /* used by libc4 */
```

.....

- 마지막에 sys_hello 함수 엔트리 추가

```
/* 320 */   .long   sys_get_mempolicy
            .long   sys_set_mempolicy
/* 322 */   .long   sys_hello

            .rept   NR_syscalls – (. – 100b) / 4
                    .long   sys_ni_syscall
           .endr
```

Makefile 수정 및 커널 재 컴파일/설치

- 추가된 hello.c가 위치한 디렉토리(kernel)의 Makefile 수정
  - obj-y에 hello.o를 추가함

```
#
# Makefile for the linux kernel.
#

obj-y     = sched.o fork.o exec_domain.o panic.o printk.o profile.o \
            exit.o itimer.o time.o softirq.o resource.o \
            sysctl.o capability.o ptrace.o timer.o user.o \
            signal.o sys.o kmod.o workqueue.o pid.o \
            rcupdate.o intermodule.o extable.o params.o posix-timers.o \
            kthread.o wait.o kfifo.o sys_ni.o posix-cpu-timers.o \
            hrtimer.o hello.o

ifeq ($(CONFIG_LTT),y)
        obj-y += ltt-base.o
endif

obj-$(CONFIG_FUTEX) += futex.o
obj-$(CONFIG_GENERIC_ISA_DMA) += dma.o
.....
```

- 커널 재 컴파일 및 타겟 시스템 설치

시스템 호출을 수행하는 사용자 프로그램 작성

- 사용자 프로그램 test-hello.c 작성

```
#include <linux/unistd.h>
#include <errno.h>
#define __NR_hello (__NR_SYSCALL_BASE+322)
/* system call stub function */
_syscall0 (void, hello);                    // hello 함수 선언 – 인수 없음

int main()
{
        hello();
        return 0;
}
```

시스템 호출 함수 선언을 위한 매크로

- unistd.h에 정의되어 있음

  syscall**x** (type, name, type1, arg1, type2, arg2, · · · , type**x**, arg**x**);

  - type: 시스템 호출 함수의 return 자료형
  - name: 시스템 호출 함수이름 (sys_를 제외)
  - type1, arg1 : 첫번째 인수의 자료형과 인수이름

    ...
  - typex, argx : x번째 인수의 자료형과 인수이름

- 커널에 시스템 호출을 추가하는 경우는 많지 않음

Handling Interrupts from Peripherals

- This requires a Linux device driver
  A. Modify the Linux kernel
  B. Write a kernel module

- Kernel modules can be loaded at run-time
  - Easily register interrupt handler using library functions

Kernel Modules for handling interrupts

- Initialization
  - Sets up the module upon loading
    module_init(init_routine);

- Interrupt handling (optional)
  - Register an interrupt handler
    request_irq( )
  - Unregister an interrupt handler
    free_irq( )

- Exit routine
  - Clears used resources upon unloading
    module_exit(exit_routine);

Exercise: Creating a Linux Device Driver

- We will use the red LEDs and pushbuttons
- The program increments the value displayed on the LEDs when a pushbutton is pressed

Kernel Module - Initialization

```
void *lwbridgebase;

static int __init intitialize_pushbutton_handler(void)
{
    // get the virtual addr that maps to 0xff200000
    lwbridgebase = ioremap_nocache(0xff200000, 0x200000);
    // Set LEDs to 0x200 (the leftmost LED will turn on)
    iowrite32(0x200, lwbridgebase); // the leftmost LED turn on
    // Clear the PIO edgecapture register(clear any pending intr)
    iowrite32(0xf,lwbridgebase+0x5c);
    // Enable IRQ generation for the 4 buttons
    iowrite32(0xf,lwbridgebase+0x58);

    (continue)
```

```
    // Register the interrupt handler.
    return request_irq(73, (irq_handler_t)irq_handler,
                IRQF_SHARED, "pushbutton_irq_handler",
                (void *)(irq_handler));
}

module_init(intitialize_pushbutton_handler);
```

Kernel Module – Interrupt Handler

```
irq_handler_t irq_handler(int irq, void *dev_id,
                           struct pt_regs *regs)
{
    // Increment the value on the LEDs
    iowrite32(ioread32(lwbridgebase)+1,lwbridgebase);
    // Clear the edgecapture register (clears current intr.)
    iowrite32(0xf,lwbridgebase+0x5c);

    return (irq_handler_t) IRQ_HANDLED;
}
```

- IRQ_HANDLED – 인터럽트가 이 디바이스에서 나올 때 return값

Kernel Module – Exit Routine

```
static void __exit cleanup_pushbutton_handler(void)
{
    // Turn off LEDs and de-register irq handler
    iowrite32(0x0,lwbridgebase);
    free_irq(73, (void*) irq_handler);
}

module_exit(cleanup_pushbutton_handler);
```

Compile a kernel module

- Makefile - 커널 모듈 컴파일을 위한 Makefile

```
obj-m += pushbutton_irq_handler.o

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

  - uname 명령어

    ```
    # uname –r                     // print kernel release
    3.13.0-00293-g1664d72
    ```

- compile

  ```
  # make          // 커널 모듈 pushbutton_irq_handler.ko 생성
  ```

Insert Newly Compiled Kernel Module into Linux

```
# insmod pushbutton_irq_handler.ko      // 커널 모듈 설치 (파일이름)
```

List All Loaded Kernel Modules

```
# lsmod                                 // 커널 모듈 설치 확인
```

Try Toggling the Pushbuttons

- 임의의 pushbutton을 누를 때마다 red LED 값이 1씩 증가함

Remove the Kernel Module

```
# rmmod pushbutton_irq_handler          // 커널 모듈 제거 (모듈 이름 사용)
# lsmod                                 // 커널 모듈 제거 확인
```